INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Causal multicasts in overlapping groups : towards a low cost approach

Achour MOSTEFAOUI
Michel RAYNAL

N° 1871
Mars 1993

PROGRAMME 1

Architectures parallèles,
Bases de données,
Réseaux et Systèmes distribués

Causal Multicasts in Overlapping Groups: Towards a Low Cost Approach*

Achour MOSTEFAOUI, Michel RAYNAL

IRISA Campus de Beaulieu - 35042 RENNES cedex - FRANCE

{mostefaoui, raynal}@irisa.fr

Programme 1, Projet ADP (Algorithmes Distribués et aPplications)

Abstract: *Concepts of group (to structure processes) and causality (to structure sendings and deliveries of messages) are of major importance in the design of distributed systems. Mixing both concepts, the ISIS system defined causal multicast in overlapping groups. This paper presents a simple and efficient protocol that implements such causal multicasts. It compares favourably with the ISIS protocol as it uses only one vector of integers (size of this vector being the total number of groups) to timestamp messages. This low cost in the size of timestamps is obtained by using (sometimes) additional resynchronization messages. It is shown that there is a trade-off between the "as early as possible delivery time" criterium and the "as small as possible timestamps size" criterium for timestamp-based protocols implementing causal order.*

Key words: causal order, communication primitive, decentralized system, distributed system, graph of groups, group, multicast, synchronous execution, vector timestamp.

Diffusions causales dans des groupes arbitraires: un protocole simple et efficace

Résumé: Le concept de groupe (qui permet de structurer les ensembles de processus) et celui de causalité (qui structure les envois et les livraisons de messages) sont deux concepts importants des systèmes répartis. A partir de ces deux concepts le système *ISIS* définit la diffusion causale dans des groupes quelconques. Cet article présente un protocole simple et peu couteux qui implémente de telles diffusions causales. Ce protocole n'utilise qu'un vecteur d'entiers (de taille égale au nombre total de groupes) pour estampiller les messages. Ce faible cout est obtenu par l'utilisation éventuelle de messages d'acquittement. Il est montré qu'il existe un compromis entre la livraison des messages au plus tot et la taille des estampilles utilisées.

Mots clés: diffusion, estampille vectorielle, exécution synchrone, graphe des groupes, groupe, ordre causal, système distribué.

1 Introduction

Since the beginning of the computer era, operating system designers have introduced and used a series of concepts in order to master the increasing complexity of upper layers applications and underlying hardwares. A lot of these concepts have been introduced to face structural issues of operating systems, applications and machines. The first one was the concept of process introduced to master the notion of sequential program activity; then concepts such as ports, gates, channels, mailboxes, etc., were introduced to solve communication related problems. One of the last concepts introduced is the group one [4,7]; a group is a set of processes implementing some service: at the application level a client addresses only an abstract service and not the individual processes that implement it.

* Supported in part by the Commission of European Communities under ESPRIT Programme Basic Reasearch Action 6360 (BROADCAST) and by the French CNRS (C^3 Project).

To face dynamic aspects of computations, concepts and mechanisms have also been introduced. For distributed systems, the *happened before* concept, introduced by Lamport [11], is one of the most outstanding ones: it allows a programmer to see an execution of her program as a partial order on events produced by this execution; this partial order expresses the *cause-effect* relation between these events. Several clock mechanisms have been introduced to express this causality relation [9,13]. In [4], Birman and Joseph introduced the concept of *causal order* that is on both groups and causality. Within a group, a message m_1 is said to precede m_2, noted $m_1 \rightarrow m_2$, if their sendings are causally related, the event *sending* of m_1 belonging to the causes of the event *sending* of m_2. Causal order is satisfied if for any couple (m_1, m_2) of such messages, that have the same destination process, the delivery of m_1 happens before the delivery of m_2.

Similar propositions have been done and implementations have been designed by several authors [5,12,15,17]. In a general framework there can be several groups (possibly overlapping) and each message is multicast within a group. Groups and multicast have been implemented in the *ISIS* system [5]. To ensure respect of causal order, this system provides a protocol that works along the following lines. Each process manages a vector of integers per group, that represents its knowledge for each member of the group, of the number of messages multicast by this member process within this group; moreover each message is timestamped with the whole set of vectors of the sending process. This information allows, when a message arrives at a destination process, to delay its delivery until the causal order is verified. This protocol is correct but expensive: each process P_i has to manage a vector per group (whether P_i belongs to this group or not) and each message has to piggyback the whole set of these vectors. This paper proposes a protocol that needs for each message the piggybacking of only one value per group, so a message carries only one vector of integers whose dimension is the number of groups. If the number of groups is large and if each of them contains a lot of processes, the saving can be very high, so the resulting protocol is interesting and practical thanks to both its simplicity and its low cost.

Section 2 gives a formal definition of causal order and states briefly the results of [5]: their protocol and their theorem about the cost that must be paid to implement causal order. Section 3 presents the low cost protocol. It is based on a synchronous approach, in some cases additional resynchronization messages are necessary to ensure liveness of deliveries. Section 4 proves the protocol, namely the safety property (causal order is ensured) and the liveness one (each message is delivered). Finally, Section 5 analyzes the trade-off between the criterium "deliver a message as soon as possible" and the criterium "size of timestamps must be as small as possible".

The proposed protocol is a step towards low-cost protocols; the search for such lightweight protocols is quite obviously a trends of future distributed computing systems. Failures are not addressed in this paper; techniques similar to the ones developed in [12,19] can be used to face these problems.

2 The causal order abstraction

2.1 Processes and groups

We consider a system composed of n processes $P_1, ..., P_n$ that communicate by exchanging messages through FIFO or non FIFO channels. There is neither common shared memory nor a global clock. Processes execute at their own speed and message transfer delay is finite but unpredictable. In other words the underlying system is reliable and asynchronous.

Cooperation between processes is structured with the group concept. Each group has a name and a set of member processes. Groups are supposed statically defined (failures and dynamic groups will be addressed in a future work). A process can belong to several groups. Let G the set of all groups and G_i the set of groups process P_i is member of. Let $GG=(V,E)$ the following undirected graph called graph of groups [5] (in a general case such a graph can have cycles):

• V is the set of groups (V=G).

• $(g_x, g_y) \in E \Leftrightarrow g_x \cap g_y \neq \emptyset$ (two groups are immediate neighbors if there is a process that is member of these two groups).

Execution of a process produces a sequence of events. Only communication events are relevant for our purpose. The multicast within a group g_x of a message m by process P_i constitutes an event noted $send_i(m)$. Every process P_j of g_x will receive this message (as channels are reliable) and the delivery of m to such a P_j constitutes an event noted $del_j(m)$; after its delivery P_j can interpret the content of m.

2.2 Causal order

Consider the events of type *send* and *del* produced by all the processes. These events are structured by the following partial order relation [11] called *causality relation* and noted $\rightarrow$:

i. for any two events e_1 and e_2 produced by the same process, if e_1 is produced before e_2 then: $e_1 \rightarrow e_2$.

ii. for any message m multicast by P_i within a group g_x ($P_i \in g_x$) and for, and only for, any $P_j \in g_x$:

   a. $del_j(m)$ is an event of P_j.

   b. $send_i(m) \rightarrow del_j(m)$.

iii. if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$.

Such a relation characterizes any distributed computation based on multicast within groups executed in an asynchronous environment. In order to be a *causal order* this relation has to satisfy the following additional constraint:

iv. if $send_i(m) \rightarrow send_j(m')$ and the messages m and m' have been multicast within respectively g_x and g_y then for any process P_k such that $P_k \in g_x \cap g_y$ we must have $del_k(m) \rightarrow del_k(m')$.

For an execution respecting causal order the notation $m \rightarrow m'$ is a shorthand for $send_i(m) \rightarrow send_j(m')$, i.e., their sendings are causally related.

2.3 Sketch of the Birman et al.'s protocol

A way to ensure that *del* events are produced according to causal order is to distinguish arrived messages and delivered messages and to delay arrived messages until some delivery condition is verified. So in the implementation there are additional *arrival* events managed by the protocol; these events are not visible at the application level.

To express delivery condition the protocol of Birman et al. uses vector clocks [5]. A vector with one entry per member process is associated to each group; the entry corresponding to some process indicates the number of multicasts executed by this process within the corresponding group. In order to ensure correct deliveries each process keeps track of a (perhaps approximate but consistent) view of the global state of the execution with regard to multicasts. This view is composed of an image of each vector, and so each process manages $|G|$ vectors (a vector per group). Moreover each message multicast within a group carries a timestamp that is a copy of the $|G|$ vectors of the sending process. This piggybacked information combined with the global state view of a receiving process allows it to express a correct delivery condition for a message that has arrived. Finally, when a message is delivered, the receiving process updates its global state view with the control information (i.e. the timestamp) carried by the deliv-

ered message.

In the protocol of Birman et al. each message carries, and each process manages $|G|$ vectors. As the protocol is correct this is sufficient to ensure causal order, but the following question rises: is it possible to design a less expensive protocol? Birman et al. provide in [5] an answer for the family of timestamp-based protocols: if the graph of groups *GG* contains cycles, the control information managed by each process and carried by each message has to concern the whole system. Intuitively a process P_i, that multicasts a message within a group g_x, has to inform destination processes about the whole subset of the causality relation it knows. In this way processes are informed of subsets of the causality relation that occurred in their back (i.e. in groups they don't belong to). Such a dissemination of information allows *del* events to be correctly produced. Figure 1.a shows the graph associated to the following groups: $g_1=\{P_1, P_2\}$, $g_2=\{P_2, P_3\}$ and $g_3=\{P_1, P_3\}$. Figure 1.b shows an execution in which message m_1 is multicast by P_1 within g_1 that is to say, as indicated by the graph of groups, in the back of P_3; m_2 is then multicast by P_1 within g_3, and m_3 is multicast by P_3 within g_2 after it was delivered m_2. To ensure causal order the delivery of m_3 (that has arrived first at P_2) has to be delayed after the delivery of m_1. This simple example shows why each process has to maintain information about multicasts in groups it does not belong to. That is necessary only if the graph of groups has cycles; if *GG* is acyclic it is sufficient for each process to manage only vectors associated to the groups it is member of; in this case a process does not need information about the subset of the causality relation it is not involved in.

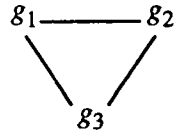

Figure 1.a. A graph of groups with a cycle

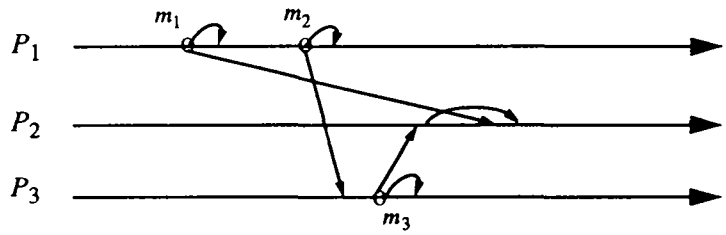

Figure 1.b. Causal delivery

3 Principle of the proposed protocol

3.1 Synchronous broadcast-based executions

Assume there is only one group (all processes receive all messages) and channels are FIFO. Consider the distributed execution displayed on Figure 2. This execution proceeds by phases; in each phase each process broadcasts exactly one message, and a phase begins only when messages sent in the previous one have been delivered; in other words the computation is synchronous [3,16(chapter 3)]. By construction we have the following, x and x' being phase numbers:

$$\forall x, x' : \forall P_i, P_j : x < x' \Rightarrow send_i(m_i^x) \rightarrow send_j(m_j^{x'})$$

and $$\forall x, x' : \forall P_k : m_i^x \rightarrow m_j^{x'} \Rightarrow del_k(m_i^x) \rightarrow del_k(m_j^{x'})$$

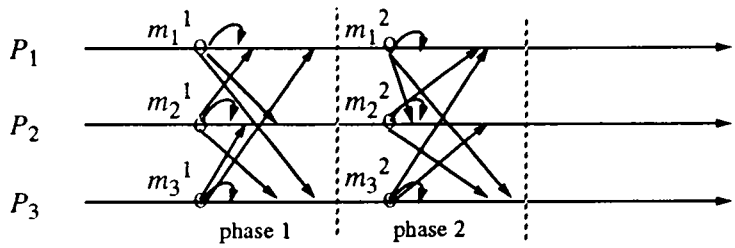

Figure 2. A synchronous execution

Causal order is naturally ensured by such a synchronous execution without additional information or control messages. The principle of the proposed protocol is to create such sequential phases at the implementation layer by timestamping messages with their phase number and delivering them according to the order on their timestamps [2]. As the sending of messages is governed by application processes that do not always progress synchronously it is possible that some P_j does not send messages during several phases. In that case such a process P_j is asked, when it receives a message timestamped t, to indicate it. To do that, P_j sends *resynch*(t+1) indicating that the next phase in which P_i will possibly send messages will not be lower than t+1; Figure 3 illustrates this rule that simulates a synchronous reset by P_2.

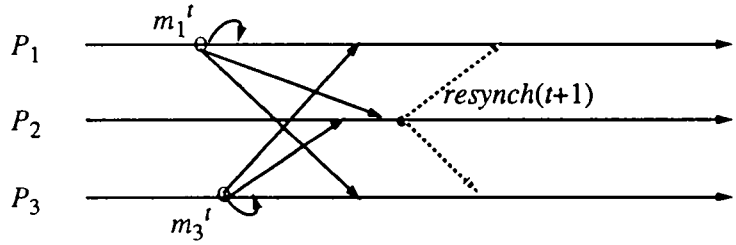

Figure 3. Simulating a synchronous behavior

Such a principle has been extensively used. For example Schneider used it in [18] (where it is called *full acknowledgment*) to express message stability. It is also used as basic mechanism in distributed discrete event simulation based on conservative method [8,10,14] (it is called here the *null message* approach). In all cases such a technique is used to ensure liveness in message delivery. A similar technique is also used to compute consistent snapshot in [6]: when a process receives for the first time a marker on an input channel, it "resets" by broadcasting markers on all its outgoing channels.

Such a use of additional *resynch* messages is actually not very expensive as a lot of transport layers use some sort of acknowledgment mechanism to ensure reliability. Moreover, if the underlying network is a communication bus, broadcasting a *resynch* message to all the other processes consists in only one operation.

3.2 Basic protocol: first step

Assuming FIFO channels (this assumption will be removed in Section 3.4), this Section presents a protocol implementing the previous principle when there is only one group including all the processes (whence the name basic protocol). To each process P_i is associated an underlying control process CTL_i implementing the rules that ensure causal order. Each CTL_i has a local array $expected_i[1..n]$ (initialized to (0, ..., 0)) where n is the number of processes constituting the only group; an entry has the following meaning:

$expected_i[j] = \alpha \Leftrightarrow$ to CTL_i's knowledge the next timestamp used by CTL_j will be greater or

equal to α.

CTL_i uses $expected_i[i]$ to timestamp messages broadcast by P_i (this variable represents CTL_i's view of the next phase number of the synchronous approach; see Section 3.1). The protocol associated to each process CTL_i is triggered by four types of events; two of them are visible by P_i (*send* and *del*; statements BS1 and BS4) while the other two are not (arrival of a message and arrival of *resynch* are visible only by the protocol; statements BS2 and BS3).

When P_i sends a message, CTL_i timestamps it with the appropriate phase number $expected_i[i]$ and sends it to each other CTL_j (statement BS1) (all these physical emissions are represented by a unique broadcast operation). When a message (m, k) arrives from P_j (statement BS2), CTL_i puts it in a set $pending_i$ (this set, initially empty, keeps messages arrived and not yet delivered) and then updates $expected_i[j]$ to $k+1$. If $expected_i[i]$ is too low it is updated and the new value is broadcast thanks to *resynch* messages. When $resynch(t)$ arrives from CTL_j (statement BS3), CTL_i updates $expected_i[j]$ accordingly. Finally (statement BS4), if for a message m timestamped k of $pending_i$ we have $min_{j=1..n}(expected_i[j]) \geq k$, that means that all messages m' sent by any P_j and such that $m' \rightarrow m$ have already arrived at CTL_i (as channels are FIFO). So the previous predicate is the delivery condition; moreover a message is delivered as soon as its delivery condition becomes true.

More formally CTL_i can be stated as follows (each statement BS1, BS2, BS3 and BS4 is executed atomically). The proof will be given for the general case in with there are several overlapping groups with possibly cycles in the graph of groups.

BS1: **When** P_i **sends** m
broadcast $(m, expected_i[i])$;
$expected_i[i]$:=$expected_i[i]$+1;
deliver m **to** P_i;

BS2: **When** (m, k) **arrives from** CTL_j
put (m, k) **in** $pending_i$;
$expected_i[j]$:=k+1;
if $expected_i[i] < k+1$ **then** $expected_i[i]$:=k+1;
broadcast $resynch(expected_i[i])$;
fi

BS3: **When** $resynch(t)$ **arrives from** CTL_j
$expected_i[j]$:=t;

BS4: **Let** (m, k) **an element of** $pending_i$
if $min_{j=1..n}(expected_i[j]) \geq k$ **then deliver** m **to** P_i;
suppress (m, k) **from** $pending_i$;
fi;
% the delivery of a message is executed as soon as its delivery condition becomes true; in an implementation it means that each time an entry of $expected_i$ is updated and $pending_i$ is not empty the previous statement is executed %

3.3 Basic protocol: final version

Phase-based synchronization and FIFO channels have very interesting consequence, that is used in the final version of the algorithm. Consider a process P_j and the sequence of the timestamps of the messages and *resynch* it sends. We have the two following properties:

i. Timestamps of consecutive messages (without *resynch* within this sequence) are consecutive integer values $\alpha - 2$, $\alpha - 1$, α, ... (see statement BS1).

ii. If at some phase β, CTL_j does not send a message, it will send *resynch*(β) (statement BS2). But it is shown in [2,3] that in that case we have necessarily as a consequence of the phase-based synchronization $\beta = \alpha + 1$, where α is the last timestamp of a message sent by P_i (Figure 4 where m stands for any particular message):

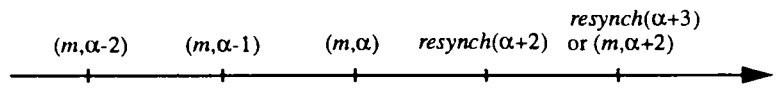

Figure 4. Timestamps of successive broadcasts

From *i*) and *ii*) it follows that timestamp values carried by consecutive messages and *resynch* sent by CTL_j obey a regular increase defined by the following rules:

- +1 between two consecutive messages
- +2 between a message and a *resynch* that follows it immediately
- +1 between to successive *resynch*
- 0 between a *resynch* and a message that follows it immediately.

As a consequence of these rules, timestamp values need not to be carried: any CTL_i can locally compute the timestamp associated to any message or *resynch* when it receives it. The protocol so obtained constitutes the basic protocol: with FIFO channels no timestamps are piggybacked.

3.4 Non FIFO channels

If channels are not FIFO it is possible for a process to simulate a FIFO arrival order (for each channel) if messages and *resynch* carry timestamps. Those timestamps are then used to trigger arrival events according to the preceding rules. In other words channels can be or not FIFO but no timestamps have to be exchanged in case of FIFO channels.

3.5 Cost of the basic protocol

In all cases, with FIFO channels, messages do not carry timestamps. If the execution is perfectly synchronous (as in Figure 2) the basic protocol has a zero cost as no *resynch* is sent; if it is not perfectly synchronous some additional *resynch* are needed to ensure liveness of deliveries. In other words the more synchronous the computation is, the more efficient the protocol is from the point of view of "as early as possible delivery occurrences". Moreover if channels are not FIFO only one integer value has to be piggybacked by messages and *resynch*, and not a vector of integers as [5] in the same situation.

As usual, timeouts can be used to decrease the number of *resynch* messages. Instead of sending a *resynch* message immediately when it receives a message (m, k) such that $expected_i[i] < k+1$, CTL_i can set a timer. Then if P_i sends a message, CTL_i switches the timer off; if the timer expires before a message is sent, CTL_i broadcasts a *resynch* message (a technique based only on local timers is used in [12]).

4 The protocol

4.1 A general protocol

To facilitate presentation, channels are assumed FIFO and messages *resynch* carry timestamps of the group they are multicast to. Results of Sections 3.3 and 3.4 can be applied to get an improved general protocol.

A set G of groups structures the set of processes. These groups can overlap and the graph of groups can be arbitrary. A *send* event by P_i is now the multicast of a message to all the members of a group to which process P_i belongs (we suppose without loss of generality there is an operation that realizes such multicasts). G_i is the set of groups P_i is member of (cf. Section 2.1). The previous basic protocol is generalized in the following way to ensure causal order. CTL_i (the basic protocol associated to P_i) is multiplexed over all groups P_i is member of, with regard to statements BS1, BS2 and BS3; so we get statements GS1, GS2 and GS3 of the general protocol. To do that CTL_i manages an array data structure $expected_i^x$ per group g_x to which P_i belongs with one entry per member of the group:

for $g_x \in G_i$, $P_j \in g_x$:
$expected_i^x[j] = \alpha \Leftrightarrow$ to CTL_i's knowledge the next timestamp used by CTL_j to multicast within g_x will be greater or equal to α.

In order to ensure correct deliveries each CTL_i manages a vector K_i of size $|G|$ (one entry per group) and uses it to timestamp messages; K_i is initialized to (0,...,0). Its meaning is the following one:

$K_i[x] = \alpha \Leftrightarrow$ α is the highest timestamp value concerning g_x and known by CTL_i.

If $P_i \in g_x$ we have $K_i[x]=expected_i^x[i]$ (but we keep the two variables to facilitate the presentation).

The delivery to P_i of a message m timestamped K is done as soon as CTL_i is sure causal order cannot be violated, that is to say when (and that gives the last statement GS4):

$\forall g_x \in G_i: \forall P_j \in g_x : expected_i^x[j] \geq K[x]$.

The four statements GS1, GS2, GS3 and GS4 executed by CTL_i can now be stated in the following way:

GS1: **When** P_i **sends** m **to** g_x
multicast (m, K_i, x) **to** g_x; 1
$expected_i^x[i]:=expected_i^x[i]+1$; 2
$K_i[x]:=expected_i^x[i]$; 3
deliver m **to** P_i; 4

GS2: **When** (m, K, x) **arrives from** CTL_j
put (m, K) **in** $pending_i$;
$expected_i^x[j]:=K[x]+1$; 5
if $expected_i^x[i]<K[x]+1$ **then** $expected_i^x[i]:=K[x]+1$; 6
$K_i[x]:=K[x]+1$; 7
multicast $resynch(K[x]+1, x)$ **to** g_x; 8
fi

GS3: **When** $resynch(t, x)$ **arrives from** CTL_j
$expected_i^x[j]:=t$; 9

GS4: **Let** (m, K) **an element of** $pending_i$
if $\forall g_x \in G_i: \forall P_j \in g_x : expected_i^x[j] \geq K[x]$ 10
then deliver m **to** P_i;
suppress (m, K) **from** $pending_i$;
$\forall g_y \notin G_i: K_i[y]:=\mathbf{max}(K_i[y], K[y])$; 11
fi;
% If the delivery conditions of several messages m_1, m_2,... become true simultaneously and if their timestamps K_1, K_2,... are ordered $K_1 < K_2 < ...$ they are delivered in the increasing timestamps order. $K_1 < K_2 \Leftrightarrow (K_1 \neq K_2$ and $\forall x: K_1[x] \leq K_2[x])$ %

If channels are FIFO *resynch* do not have to carry timestamp and a message multicast has not to carry the field $K[x]$, as these ones can be processed by destination processes (Section 3.3). If channels are not FIFO these timestamps have to be carried (Section 3.4).

4.2 An example

Three processes P_1, P_2 and P_3 are structured with three groups $g_1=\{P_1, P_2\}$, $g_2=\{P_2, P_3\}$ and $g_3=\{P_1, P_3\}$ (Figure 1.a displays the corresponding graph of groups). Messages are multicast in the following way: m_1 by P_1 within g_1, m_2 by P_1 within g_3 and m_3 by P_3 within g_2, with the following causal relation: $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$ so m_1 must be delivered to P_2 before m_3 to ensure causal delivery. In Figure 5, K_i appears as $K_i[g_1, g_2, g_3]$.

- denotes a *send* event.
- □ denotes a *del* event.

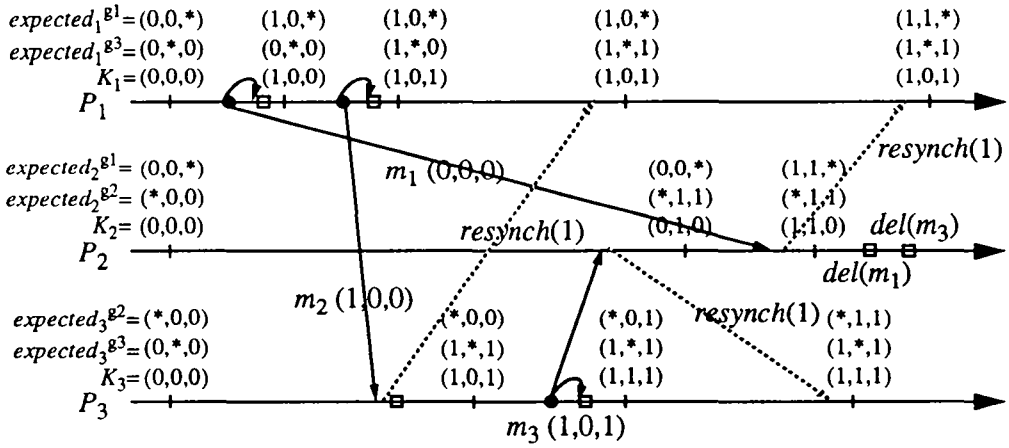

Figure 5. An example of causal delivery.

4.3 Particular case

If there is no cycle in the graph of groups, each vector K_i can be reduced to only entries $K_i[x]$ for groups $g_x \in G_i$ (in other words, as for these groups $K_i[x]=expected_i^x[i]$, array K_i is no longer necessary). As in [5] a process then manages only control information related to groups it belongs to.

4.4 Proof

We assume without loss of generality that channels are FIFO (cf. Section 3.4).

4.4.1 Proof of the safety property

The safety property states that causal order is never violated. To establish this result two lemmas about monotonicity properties are first proved.

Lemma 1: Monotonic behavior of vectors K_i. The successive values of each local vector K_i are monotonically increasing.

Proof of lemma 1: This follows directly from lines 2, 3, 6, 7 and 11 of the protocol. ❑

Lemma 2: Timestamps of causally ordered messages are monotonically increasing. Let m_1 and m_2 two messages timestamped $K(m_1)$ and $K(m_2)$, respectively sent by P_i and P_j; m_1 has been sent to group g_{x1}. We have: $m_1 \rightarrow m_2 \Rightarrow K(m_1) \leq K(m_2)$ and $K(m_1)[x_1] < K(m_2)[x_1]$.

Proof of lemma 2:

- Case 1: $i=j$ (m_1 and m_2 have been sent by the same process). After the sending of m_1 to g_{x1}, CTL_i increments $K_i[x_1]$ (lines 2, 3) and consequently, as the sending of m_2 occurs after the one of m_1, $K(m_1) \leq K(m_2)$ and $K(m_1)[x_1] < K(m_2)[x_1]$.

•Case 2: $i \neq j$. We consider two subcases.

Case 2.1: P_j belongs to g_{x1} the group destination of m_1; in other words:

$$send_i(m_1) \rightarrow del_j(m_1) \rightarrow send_j(m_2)$$

After the arrival of m_1 to P_j we have (lines 5, 6 and 7) $K_j[x_1] > K(m_1)[x_1]$. Moreover after the delivery of m_1 to P_j we have:

$\forall g_y \in G_j$: $expected_j^y[j] \geq K(m_1)[y]$ i.e. $K_j[y] \geq K(m_1)[y]$ (line 10)
$\forall g_y \notin G_j$: $K_j[y] \geq K(m_1)[y]$ (line 11)

Consequently after the delivery of m_1 to P_j we have $K_j \geq K(m_1)$ and $K_j[x_1] > K(m_1)[x_1]$. It follows that at the sending of m_2, as the value of K_j do not decrease (lemma 1) we have $K(m_2) \geq K(m_1)$ and $K(m_2)[x_1] > K(m_1)[x_1]$.

Case 2.2: P_j does not belong to g_{x1} the group destination of m_1; $m_1 \rightarrow m_2$ implies that there is a finite sequence of messages m_1, m', m'' ,..., m''' , m_2 such that:

$$send_i(m_1) \rightarrow del_a(m_1) \rightarrow send_a(m') \rightarrow del_b(m') \rightarrow send_b(m'') \rightarrow \ldots \rightarrow del_j(m''') \rightarrow send_j(m_2)$$

by applying the result of case 2.1 to the sequence of couples (m_1, m'), (m', m'),..., (m''', m_2) we get $K(m_1) < K(m') < K(m'') < \ldots < K(m''') < K(m_2)$, and that proves lemma 2. ❑

Proof of safety: To show causal order is never violated we consider the two following cases.

•Case 1: P_i can immediately deliver a message m_2 it has sent without violating causal order (line 4).

Consider first $m_1 \rightarrow m_2$ and both m_1 and m_2 have been sent by P_i: then according to delivery rule m_1 has been delivered; second, if m_1 has been sent by another P_j, definition of $m_1 \rightarrow m_2$ indicates delivery of m_1 occurs before the sending of m_2.

•Case 2:

if: • P_i is a destination process for m_1 and m_2
• $m_1 \rightarrow m_2$
• m_2 has arrived and is deliverable

then m_1 has also arrived at P_i and is deliverable (and then it is deliverable before m_2 as it has a lower timestamp, see GS4).

Let $K(m_1)$ and $K(m_2)$ the timestamps of m_1 and m_2. If m_2 is deliverable we have (line 10):

$$\forall g_x \in G_i : \ \forall P_j \in g_x : \ expected_i^x[j] \geq K(m_2)[x]$$

As $\forall g_x \in G : \ K(m_2)[x] \geq K(m_1)[x]$ (lemma 2) and variables $expected_i^x[j]$ never decrease the delivery condition of m_1 is also true.

We show now that m_1 has arrived at P_j. Let P_{j1} and g_{x1} be respectively the sender and the group destination of m_1. As $m_1 \rightarrow m_2$ by lemma 2, $\forall g_x \in G : \ K(m_2)[x] \geq K(m_1)[x]$ and $K(m_2)[x_1] > K(m_1)[x_1]$. P_i is member of g_{x1} (as it is a destination process of m_1). As the delivery condition of m_2 is true we conclude: $expected_i^{x1}[j_1] \geq K(m_2)[x_1] > K(m_1)[x_1]$.

The variable $expected_i^{x1}[j_1]$ can increase only when P_i receives a message $(m, K(m), x_1)$ (line 5) or $resynch(t, x_1)$ (line 9) from CTL_{j1}. It follows that P_i has received from CTL_{j1} a message m with $K(m)[x_1] \geq K(m_1)[x_1]$ or $resynch(t)$ with $t > K(m_1)[x_1]$, that updated $expected_i^{x1}[j_1]$ to its current value:

• if it has received a message m with $K(m)[x_1]=K(m_1)[x_1]$ then m is m_1 (from lemma 1).

• if it has received a message m with $K(m)[x_1] > K(m_1)[x_1]$ then $m_1 \rightarrow m$ and as channels are FIFO as m has arrived m_1 has also arrived.

• if it has received $resynch(t, x_1)$, all the messages m sent by P_{j1} to P_i before this *resynch* message had timestamps lower than t (m_1 is among them) and have already arrived as channels are FIFO.

Consequently in all cases m_1 has already arrived. ❑

4.4.2 Proof of the liveness

The liveness property states that each message multicast within a group will eventually be delivered once at each process of the destination group.

Proof of liveness: A message sent by P_i is delivered to it once (line 4). Now consider the case of a message (m, K) that arrived and has been put in $pending_i$. If its delivery condition is false we have (line 10): $\exists g_x \in G_i$, $\exists P_j \in g_x$: $expected_i^x[j] < K(m)[x]$.

Consider such a couple (g_x, P_j) and the two following cases (P_i and P_j belong to g_x).

Case 1: $expected_j^x[j] \geq K(m)[x]$. This means that CTL_j sent within g_x a message m_1 timestamped $K(m_1)$ with $K(m_1)[x] \geq K(m)[x]$-1 (lines 1, 2 and 3) or $resynch(t, x)$ with $t \geq K(m)[x]$ (lines 6, 7 and 8 for P_j). As channels are reliable, data and resynchronization messages arrive at their destinations. At their arrival CTL_i updates $expected_i^x[j]$:

- to $K(m_1)[x]+1$ in the case message m_1 was sent (line 5)
- to t in the case a *resynch* message was sent (line 9)

In both cases the new value of $expected_i^x[j]$ will be greater or equal to $K(m)[x]$.

Case 2: $expected_j^x[j] < K(m)[x]$. From the monotonicity of $expected_j^x[j]$ and the test in line 6, CTL_j has not yet received m. As channels are reliable m will eventually arrive at CTL_j. When m arrives, CTL_j updates $expected_j^x[j]$ to $K(m)[x]+1$ (line 6) and sends $resynch(K(m)[x]+1, x)$ to CTL_i (line 8); and we are now in case 1.

So for each couple (g_x, P_j) we will eventually have in both cases $expected_i^x[j] \geq K(m)[x]$ and then the delivery condition for m will be true.

That proves each message is eventually delivered. Moreover as a delivered message is suppressed from $pending_i$ it is delivered once. ❑

5 About the additional cost to ensure causal order

5.1 Delivery time

In [5,17] the delivery condition of a message m arrived at P_i evaluated to false indicates that some messages that were sent to P_i and that are causally ordered before m have not yet arrived at P_i. In other words the delivery of a message is delayed if, and only if, some not yet arrived messages have to be delivered before it. In that sense the protocols of [5,17] are optimal. The proposed protocol can delay the delivery of a message for the same reason but also possibly because *resynch* messages have not yet arrived. When the distributed execution is synchronous the protocol is optimal (as in that case *resynch* messages are not used). The more synchronous the execution is, the nearest of optimal occurrences of deliveries are.

5.2 Size of timestamps and local memory cost

To ensure causal order, in spite of cycles in the graph of groups, each message has to carry some global information concerning causal order. In [5] each process P_i manages a vector K_i^x per group g_x whether P_i belongs to this group or not. Moreover each message carries as a timestamp the whole set of vectors of its sender, i.e., $|G|$ vectors. In other words the size of each timestamp is $\sum|g_x|$ (for $g_x \in G$) (counting the different fields of each timestamp). So the protocol of [5] ensures optimality with respect to delivery time (deliveries occur as early as possible) thanks to a high cost in the size of timestamps and local memory overhead.

In the proposed protocol each message is timestamped by only one vector of size $|G|$. Moreover each process P_i manages only a vector $expected_i^x$ per group to which it belongs and an additional vector for groups to which it does not belong; this additional vector is of size $|G - G_i|$ (as $K_i[x]=expected_i^x[i]$

for $g_x \in G_i$). This saving results from the approach taken (that maps a distributed execution into a virtually synchronous one [2,3]); this approach needs in some cases additional *resynch* messages (that can delay deliveries occurrences), but these *resynch* messages are only sent within the concerned group.

Another approach could be to use only one integer to timestamp messages independently from the group they are multicast to. In that case a process receiving a message m multicast to g_x would have to broadcast, if necessary, a *resynch* message to all the groups it belongs to and not only to the group g_x in order to ensure liveness of deliveries.

As we can see there is a trade-off between optimality with respect to the criterium "deliver messages as soon as possible" (i.e. no additional delay has to be added by the protocol) and optimality with respect to the criterium "use timestamps of the smallest possible size without increasing the number of resynchronization messages used". In a real system the choice of a particular protocol actually depends on several factors: number of groups, synchronism of upper layers applications, presence of an acknowledgment mechanism in the transport layer, presence of multicast buses, etc.

Conclusion

This paper has presented a simple low cost (in size of timestamps) protocol that ensures causal order between send and delivery events within overlapping multicast groups. This has been achieved by using vector of integers to timestamp messages; the size of such a vector being $|G|$ (the total number of groups). This improvement concerning timestamps size, when compared to solutions that use a vector of vectors (one vector per group of G) to timestamp each message, has been made possible thanks to additional resynchronization messages. It has been shown that there is a trade-off between as early as possible delivery time and the size of additional information messages have to piggyback. All these protocols characterize a family of protocols that ensure causal order (those that are timestamp-based). The choice of a particular protocol for a real system depends on the "optimality" its designer is looking for.

This work is being pursued in two directions. The first one consists in bounding the values domain of variables $expected_i^x[j]$ and $K_i[y]$. The second one is to resist failures of processes and to allow them to join or to leave dynamically a group: this needs to add an underlying membership algorithm [1] in order to obtain dynamic group management.

Bibliography

[1] Y. Amir, D. Dolev, S. Kramer, D. Malki
Membership algorithms for multicast communication groups.
Proc. 6th Int. Workshop on Dist. Alg., Springer-Verlag, LNCS 647, (Segall, Zacks Ed.), Haifa, (1992), pp. 292-312.

[2] M. Adam, Ph. Ingels, M. Raynal
The meaning of synchronous distributed algorithms run on asynchronous distributed systems.
Proc. 3rd Int. Symposium on Comp. and Inf. Sciences, Izmir, (1988), pp. 301-316.

[3] B. Awerbuch
Complexity of network synchronization.
Journal of the ACM, Vol. 32,4, (1985), pp. 801-823.

[4] K. Birman, T.A. Joseph
Reliable communication in the presence of failures.
ACM TDCS, Vol. 5,1, (Feb. 1987), pp. 47-76.

[5] K. Birman, A. Schiper, P. Stephenson
Lightweight and atomic group multicast.
ACM TOCS, Vol. 9,3, (1991), pp. 272-314.

[6] K.M. Chandy, L. Lamport
Distributed snapshots: determining global states of distributed systems.
ACM TCDS, Vol. 3,1, (1985), pp. 63-75.

[7] D. Cheriton, W. Zwaenepoel
Distributed process groups in the V kernel.
ACM TOCS, vol. 3,2, (1985), pp. 97-107.

[8] R.C. De Vries
Reducing NULL messages in Misra's distributed discrete event simulation method.
IEEE Trans. on Soft. Eng., Vol. 16,1, (1990), pp. 82-91.

[9] C.J. Fidge
Logical time in distributed computing systems.
IEEE Computer, Vol. 24,8, (1991), pp. 11-76.

[10] R.M. Fujimoto
Parallel discrete event simulation.
Comm. ACM, Vol. 33,10, (1990), pp. 31-53.

[11] L. Lamport
Time, clocks and the ordering of events in a distributed system.
Comm. ACM, Vol. 21,7, (july 1978), pp. 558-565.

[12] R.A. Macedo, P. Ezhilchelvan, S.K. Shrivastava
Implementing robust multicast protocols using causal blocks.
Research Report, Univ. of Newcastle Upon Tyne, (1992), 20p.

[13] F. Mattern
Virtual time and global states of distributed systems.
Proc. of Int. Workshop on Parallel and Dist. Systems, North-Holland, 1988, pp. 215-226.

[14] J. Misra
Distributed discrete event simulation.
ACM Computing Surveys, Vol. 18,1, (1986), pp. 39-65.

[15] L.L. Peterson, N.C. Bucholz, R. Schlichting
Preserving and using context information in interprocess communication.
ACM TOCS, Vol. 7,3, (1989), pp. 213-246.

[16] M. Raynal, J.M. Hélary
Synchronization and control of distributed systems and programs.
Wiley & sons, Series in Parallel Programming, (1990), 124p.

[17] M. Raynal, A. Schiper, S. Toueg
The causal order abstraction and a simple way to implement it.
Inf. Proc. Letters, Vol. 39, (1991), pp. 343-350.

[18] F.D. Schneider
Synchronization in distributed programs.
ACM TOPLAS, Vol. 4,2, (1982), pp. 125-148.

[19] P. Verissimo, L. Rodriguez, J. Rufino
The atomic multicast protocol (AMp).
In Delta-4: A generic Arch. for Dependable Dist. Comp., Springer-Verlag, Research Report ESPRIT, (D. Powell Ed.), (1992), pp. 268-293.

PUBLICATIONS INTERNES 1993

Derniers titres parus

PI 704 UNE ARCHITECTURE POUR L'EXECUTION D'APPLICATIONS SIGNAL
Krzysztof WOLINSKI, Thierry CHOLET, Dominique DERRIEN, *
Jean-Pierre GUILLOU, Paul LEALI, Alain RIBOULT
Février 1993, 30 pages.

PI 705 SIRENE-F : SYSTEME POUR LA CONSTRUCTION ET L'EXECUTION DE RESEAUX NEURO-FLOUS EN VUE D'APPLICATIONS EN TEMPS REEL
Krzysztof WOLINSKI, Pierre-Yves GLORENNEC
Février 1993, 14 pages.

PI 706 TERMINATION DETECTION IN A VERY GENERAL DISTRIBUTED COMPUTING MODEL
Jerzy BRZEZINSKI, Jean-Michel HELARY, Michel RAYNAL
Février 1993, 16 pages.

PI 707 AN ARCHITECTURE FOR TOLERATING PROCESSOR FAILURES IN SHARED-MEMORY MULTIPROCESSORS
Michel BANATRE, Alain GEFFLAUT, Philippe JOUBERT,
Christine MORIN
Février 1993, 34 pages.

PI 708 SPAM : A MULTIPROCESSOR EXECUTION DRIVEN SIMULATION KERNEL
Alain GEFFLAUT, Philippe JOUBERT
Février 1993, 22 pages

PI 709 REGRESSOR SELECTION AND WAVELET NETWORK CONSTRUCTION
Qinghua ZHANG
Février 1993, 20 pages.

PI 710 CAUSAL MULTICASTS IN OVERLAPPING GROUPS : TOWARDS A LOW COST APPROACH
Achour MOSTEFAOUI, Michel RAYNAL
Mars 1993, 14 pages.

Unité de Recherche INRIA Rennes
IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399